RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 9: Recursion, Binary Search Trees

Lecture Slides

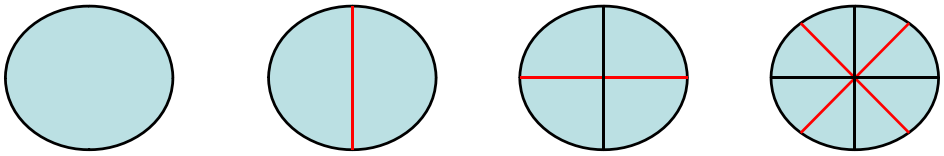
Recursion

Recursion is the process of defining something in terms of itself. The aim is to continually define a problem in terms of a simpler (or partial) form of itself, until the result is simple enough to be directly evaluated (e.g. without iteration).
The result then passes back through the previous definitions, with the answer being built up at each stage. By the time the result falls back to the original problem, the proper solution has been evaluated.

Recursion: A Simple Conceptual Example

Consider the process of cutting a cake into equal sized slices.

First, the cake is cut in two. Then, each half is cut into quarters. Each quarter is cut into two eighths .. (etc.)
This process continues until the desired slice size is obtained.



This cake cutting was a recursive process: the only task that is being done is cutting in half. The cake was first cut in half, then the halves were cut in half, and so on.

Parts of a Recursive Algorithm

Recursive algorithms are defined by two kinds of conditions:

- a recursive step
- a terminating condition

In the cake example, the *recursive step* was the act of cutting each cake section in half.
The *terminating condition* was to stop cutting when the desired cake slice size was reached.

It is possible that there may be many recursive steps and terminating conditions for a given algorithm.

A Mathematical Example (1)

A factorial of a number *x* is the product of all values from 1 up to and including *x*.
For example:

$4! = 4 \times 3 \times 2 \times 1 = 24$

(an exclamation mark is the symbol for factorial.)

The conventional iterative form of the operation is:

$x! = x \times x\text{-}1 \times x\text{-}2 \ldots \times 1$

This can also be expressed recursively as:

$x! = x \times (x\text{-}1)!$

A Mathematical Example (2)

We could use this recursive definition to calculate the result of 4 factorial:

$4! = 4 \times 3!$

And we could then use the same definition to complete the calculation:

$3! = 3 \times 2!$

$2! = 2 \times 1!$

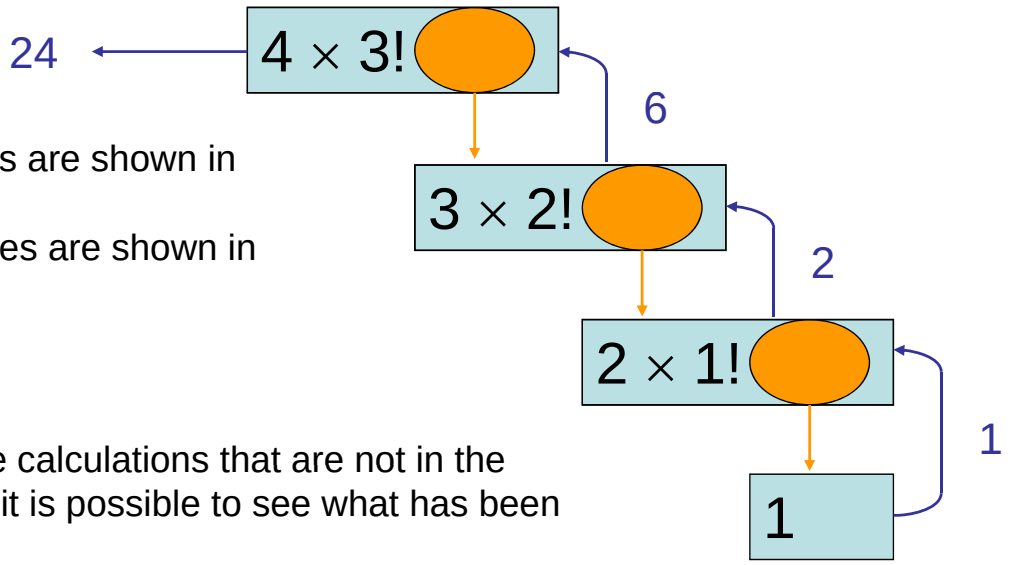
Recursive Step:
$x! = x \times (x\text{-}1)!$
Terminating Condition:
$x = 1$

1 factorial is simply 1 – it is the terminating condition (there are no more recursive steps required to arrive at a complete solution):

$1! = 1$

A Mathematical Example (3)

It is easier to see how the recursion works to arrive at the result diagramatically:



Recursive calls are shown in orange.
Returned values are shown in blue.

By reading the calculations that are not in the orange circle, it is possible to see what has been calculated.

A Mathematical Example in Java Code

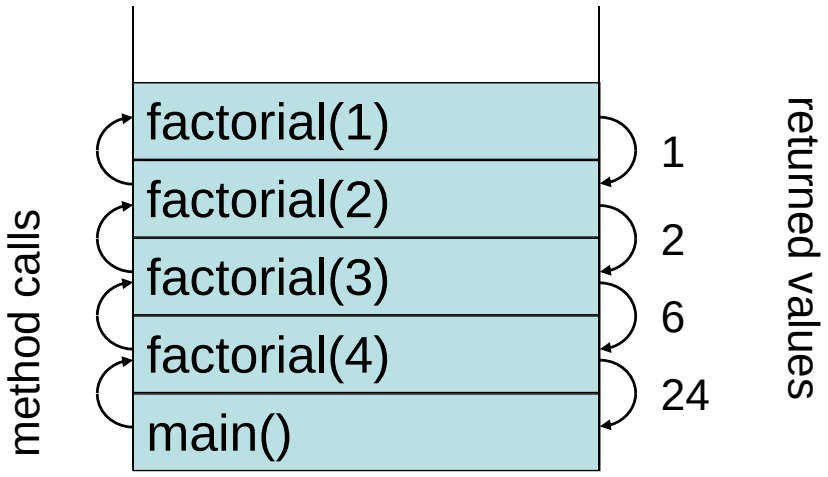
Once the recursive and terminating steps have been defined, an implementation is fairly easy to write:

```
public int factorial(int a) {
  if (a == 1)
    return 1;
  else
    return (a * factorial(a-1));
}
```

The terminating condition is highlighted in red, and the recursive step in blue.

Recursion and the Method Stack

Each recursive call to a method is placed on top of the method stack.
Calculating 4 factorial with the given source code can be visualised as:



Recursion vs. Iteration

Both recursion and iteration have their benefits and disadvantages.

Recursion is commonly found in a number of mathematical situations. It can also be a useful technique for the design of particular algorithms to manage strings, lists and other types of data structures.
Iteration is more efficient and preferable when there is a simple iterative (loop based) solution for a problem (for example, the simple factorial operation used as an example for recursion could be implemented more efficiently using iteration.)

Recursion should also be avoided for long calculations where the method stack may overflow.

Recursion vs. Iteration: Efficiency

A Fibonacci number is the sum of the previous two Fibonacci numbers. For example, 1,1,2,3,5,8,13,21,34,55,89, …

```
public static long fibonacci(int n)
{
  if (n <= 1)
    return 1;
  else
    return (fibonacci (n – 1) + fibonacci (n – 2);
}
```

```
int fib1 = 1, fib2 = 1, temp=0;
for (int i = 1; i < n; i++)
{
  temp=fib2;
  fib2=fib1+fib2;
  fib1=temp;
}
return fib2;
```

In this case, the iterative solution is more efficient than the recursive one.
Some algorithms lend themselves to iteration, others to recursion.

Thinking Recursively: String Length

The length of a string is 1 plus the length of the rest of the string.
The length of a string is 0 when the string is empty.

Recursive Step: if the string isn't empty, length = 1 + length of the remainder of the string.
Terminating condition: if the string is empty, the length is 0.

```
static int stringLen(String s)
{
  if (s.equals(""))
    return 0;
  else
    return (1 + (stringLen(s.substring(1))));
}
```

Thinking Recursively: String Reversing (1)

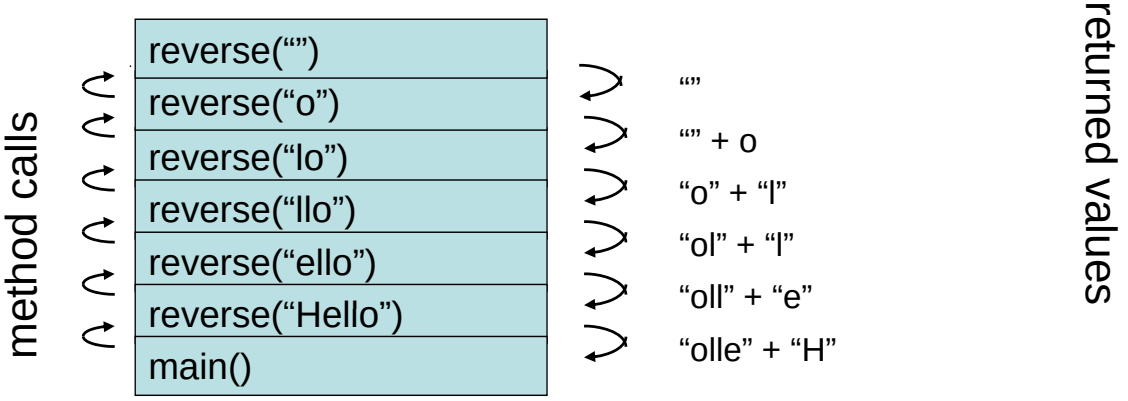
A reversed string is the reverse of all characters except the 1st, with the first character appended to the end.
A reversed string is empty when the string is empty.

Terminating condition: if the length is zero, return a blank string.
Recursive Step: return the reverse of the String (from the second character onwards), with the first character appended to the end of the result.

```
static String stringRev(String s)
{
    if (s.length()==0)
        return ("");
    else
        return (stringRev(s.substring(1)) + s.charAt(0));
}
```

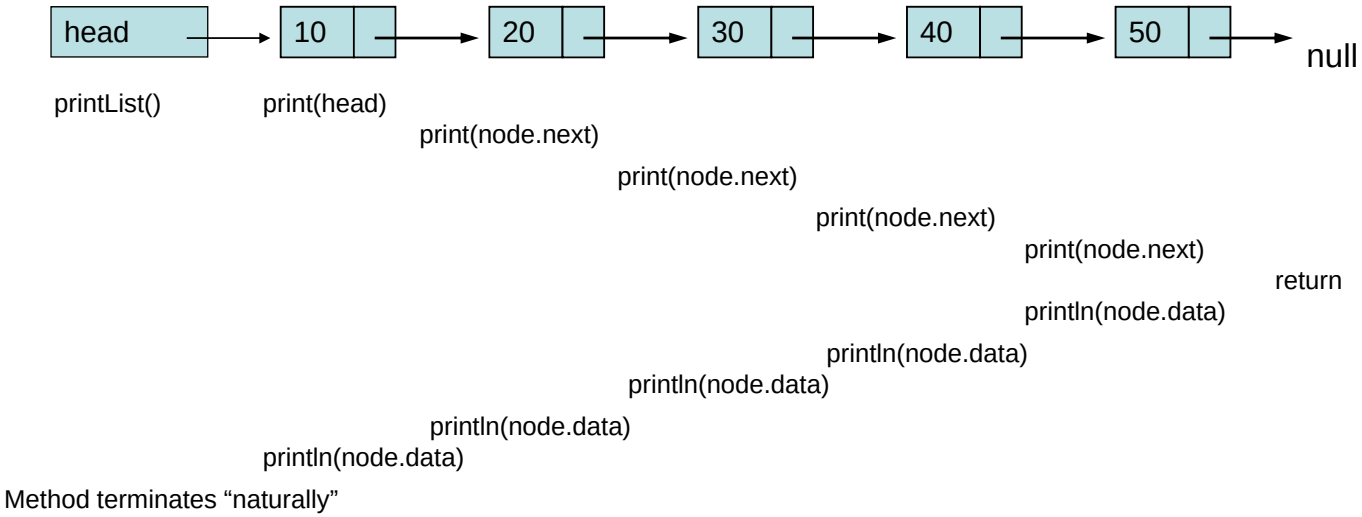

Thinking Recursively: String Reversing (2)

Again each recursive call to a method is placed on top of the method stack.
Calculating 4 factorial with the given source code can be visualised as:



Thinking Recursively: Linked List (1)

Let's look at another example – printing a linked list in reverse order.
The terminating condition will be when we reach the NULL value at the end of the list and the recursive call will effectively move to the next "node" in the list.
Once we have reached the end of the list we can print the values stored in the nodes we have visited.



Thinking Recursively: Linked List (2)

We still want to keep the details of now the data is stored in the ADT hidden, so what we have to do is add a method to the public interface for printing the list in reverse and then have this public interface method call our private recursive print method (which effectively remains hidden from the client program).

```
private void printReverse(ListNode head)
 {
   if (head != null)
   {
     System.out.println(head.number);
     printReverse(head.next);
   }
 }

private void print(ListNode head)
 {
   if (head != null)
   {
     print(head.next);
     System.out.println(head.number);
   }
 }
```

Stop calling method recursively when end of list is reached.

Print data in current node before subsequent recursive calls have been completed

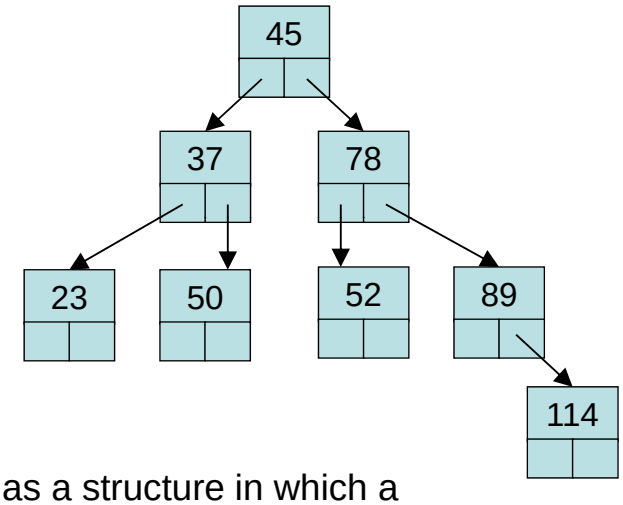
Call method again to move to next node in list

Print data in current node after subsequent recursive calls have been completed

Tree Structures

Linked lists, vectors, stacks and queues are linear (sequential) data structures. A tree is different; it is non-linear, or hierarchical.

The tree structure is naturally suited to many applications (e.g. databases, file systems, web sites, etc.), and can often allow algorithms to be significantly more efficient.



A tree (in data structure terms) is defined as a structure in which a set of *nodes* are connected together by their *edges*, in a parent-child relationship.

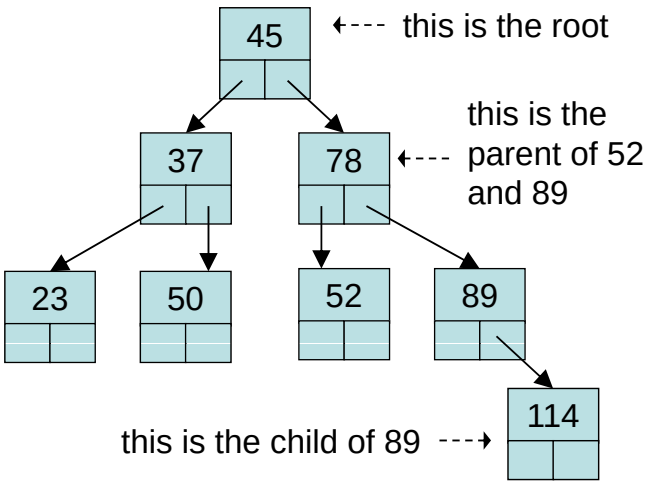
(Visualising the 'tree' relationship may be easier looking at the diagram upside-down, with the root at the bottom.)

Tree Terminology (1)

A **tree** has a single distinguished node called the *root*, and every other node is connected to only one parent (c.f. a graph).

A **binary tree** is one in which each node has at most two children, each called the *left* and *right* child.

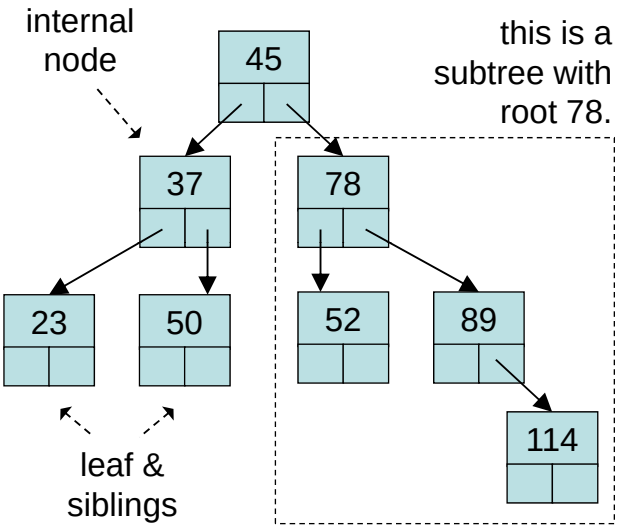
The diagram to the right is an example of a binary tree.



A recursive definition of a tree is as follows; a tree is either empty, or consists of a root node, with zero or more *subtrees* as children (read on.)

Tree Terminology (2)

- nodes with the **same parent** are called **siblings (e.g. 37 & 78)**
- an **leaf (external, or terminal) node** has **no children (e.g. 23 & 50)**
- an **internal (non-terminal) node** has **one or more children (e.g. 37 & 78)**
- if there is a path from node A to node B, then A is an **ancestor** of B, and B is a **descendant** of A.

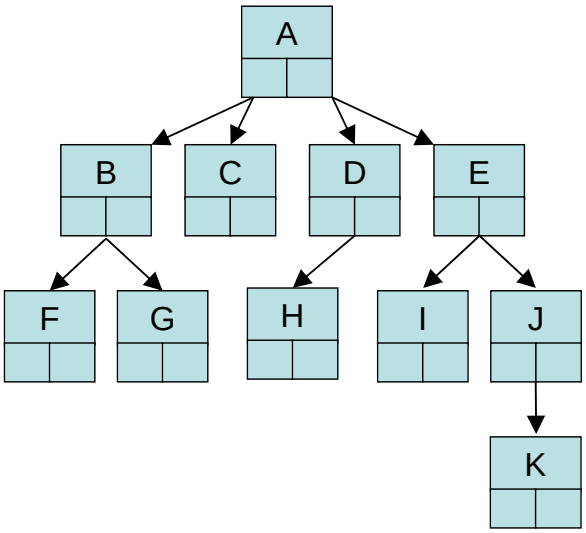


A **subtree** of a tree, rooted at a particular node, is a tree consisting of that node and all its descendants.

This reveals that a tree is inherently a **recursive structure**.

Tree Properties

- the **depth** of a node is the length of the path from the root to the node. (The root has a depth of 0.)
- the **height** of a node is the length of the path from the node to the deepest leaf.
- the **size** of a node is the number of nodes in the subtree rooted at that node (including itself.)



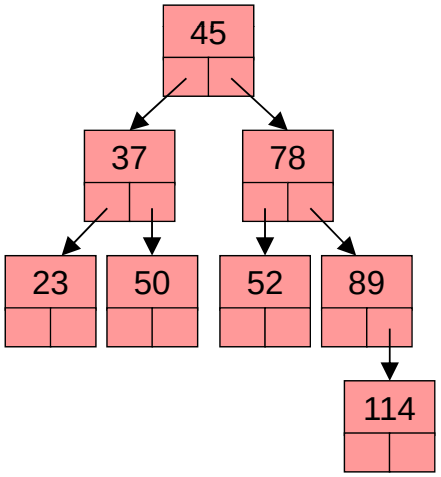
	A	B	C	D	E	F	G	H	I	J	K
Depth	0	1	1	1	1	2	2	2	2	2	3
Height	3	1	0	1	2	0	0	0	0	1	0
Size	11	3	1	2	4	1	1	1	1	2	1

Binary Search Tree (1)

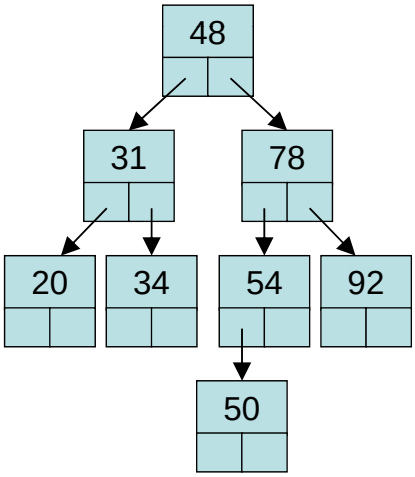
A **binary tree** is any tree with at most two descendents per node

A **binary search tree** is a tree (not automatically balanced) in which:

- all descendants **to the left** of any node have **lesser** data values
- all descendants **to the right** have **greater** data values.



not a binary search tree



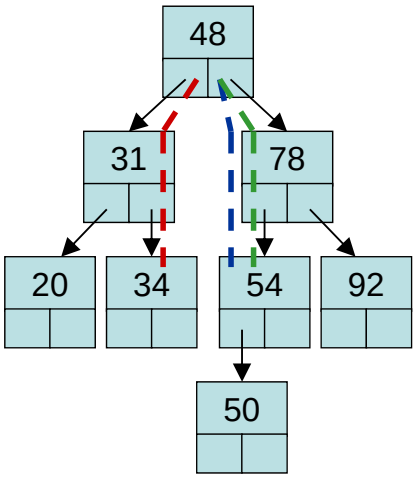
a binary search tree

Binary Search Tree (2)

A binary search tree satisfies the search order property, and can be seen as **an extension of the binary search algorithm** to allow insertions and deletions.

Best case is ***O(log N)*** as in binary search (when the tree is balanced)

Worst case is ***O(N)*** as in linear search (when the tree is linear, like a linked list)



to search for 34, we need to go only through nodes 48, 31 and 34.

to insert 60, we need to go only through nodes 48, 78 and 54.

to remove 54, we need to through nodes 48, 78 and 54. But, we also need to **reattach the tree** to maintain the ordering!

Binary Search Tree ADT (1)

Let us now implement an Abstract Data Type (ADT) for a binary search tree, and go through the algorithms for inserting, finding, traversing and removing elements.

First we must identify the basic operations to create the interface:

- **find** – find a node in the tree
- **insert** – insert a node into the tree
- **remove** – remove a node from a tree
- **printPreorder** – traverse and print every node in *pre-order*
- **printInorder** – traverse and print every node in *in-order*
- **printPostorder** – traverse and print every node in *post-order*

Binary Search Tree ADT (2)

Nodes in a binary search tree ADT must encapsulate data values of a generic type. But, **which generic class to use?**

Every class is a descendant of **`java.lang.Object`**, however unlike stacks or queues, where we were simply storing and removing elements, **we need to do comparisons** on the data contained inside the nodes.

We need a generic class that can be **compared in three ways** (equals, less than, and greater than) – one such interface is **`java.lang.Comparable`**. This method includes a method called **`compareTo()`**, which can be used for such comparisons.

Therefore, any object stored in the BST **will need to implement the `Comparable` interface.** Another restriction is that the BST's methods must use only parameters of, and return, **`Comparable`** data types.

The user of the BST **must use an object that implements `Comparable`** that is appropriate for their application.

NOTE: See refactored topic9 source code for example of using parameterised generic type and improved encapsulation

Binary Search Tree ADT Specification (1)

First, we define the interface generic to all types of binary tree.

```
interface BinaryTree
{
   public Comparable find (Comparable target);
   public boolean insert (Comparable item);
   public boolean remove (Comparable item);
   public void printPreorder ();
   public void printInorder ();
   public void printPostorder();

}
```

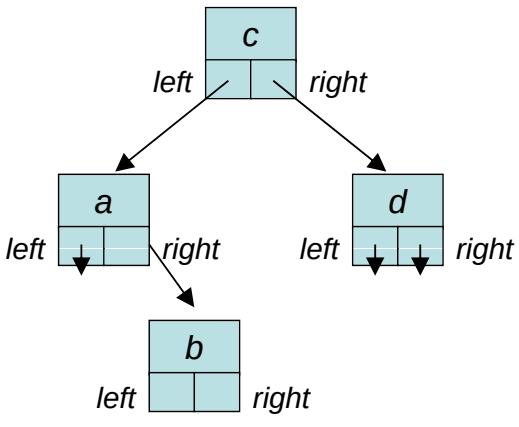
The implementation can be private since the user does not need to see it in order to use the specified methods.

Binary Search Tree ADT Node (1)

Recall the implementation of a linked list…

A tree has nodes, similarly to a linked list.

However, unlike a linked list, a tree has more than one 'next' reference; in the case of a binary tree, two references: left and right child node.



Also, recall that we need to use the interface *Comparable* for the type of the data value in the node (*see next slide.*)

Binary Search Tree ADT Node (2)

```
class BinaryNode
{
   Comparable data; // see Caspar's refactored BinaryNode
   BinaryNode left; // for better encapsulation/generics
   BinaryNode right;

   public BinaryNode(Comparable item)
   {
      data = item;
      left = null;
      right = null;
   }
}
```

Initialise the two child references to point nowhere (no children.)

Class BinaryNode and its attributes are package visible (default modifiers), so that the binary tree implementation (presumably in the same package) can use it. It is not made public, since it should only be accessible by other tree classes. (c.f. Caspar's refactored topic 9 code which makes the node class private)

Binary Search Tree ADT Public Implementation (1)

```
public class BinarySearchTree implements SearchTree
{
   protected BinaryNode root;

   public Comparable find(Comparable target) throws
                                                ItemNotFound
   {
      return find(target, root);
   }

   public void insert (Comparable item) throws DuplicateItem
   {
      root = insert (item, root);
   }

   public void remove (Comparable item) throws ItemNotFound
   {
      root = remove(item, root);
   }
```

These are calls to protected internal methods that we will define shortly.

Binary Search Tree ADT Public Implementation (1)

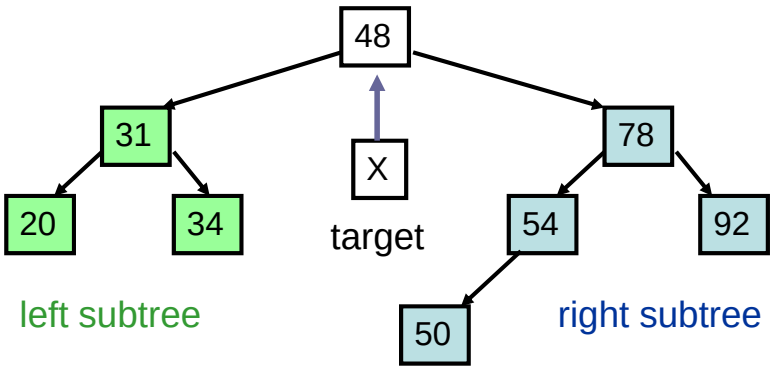
```
    public void printPreorder()
    {
       preorder(root);
    }

    public void printInorder()
    {
       inorder(root);
    }

    public void printPostorder()
    {
       postorder(root);
    }
}
```

These are calls to protected internal methods that we will define shortly.

All these methods rely on internal methods to actually do the work. We will now investigate what these internal methods have to do with the tree structure, and implement them in Java code.

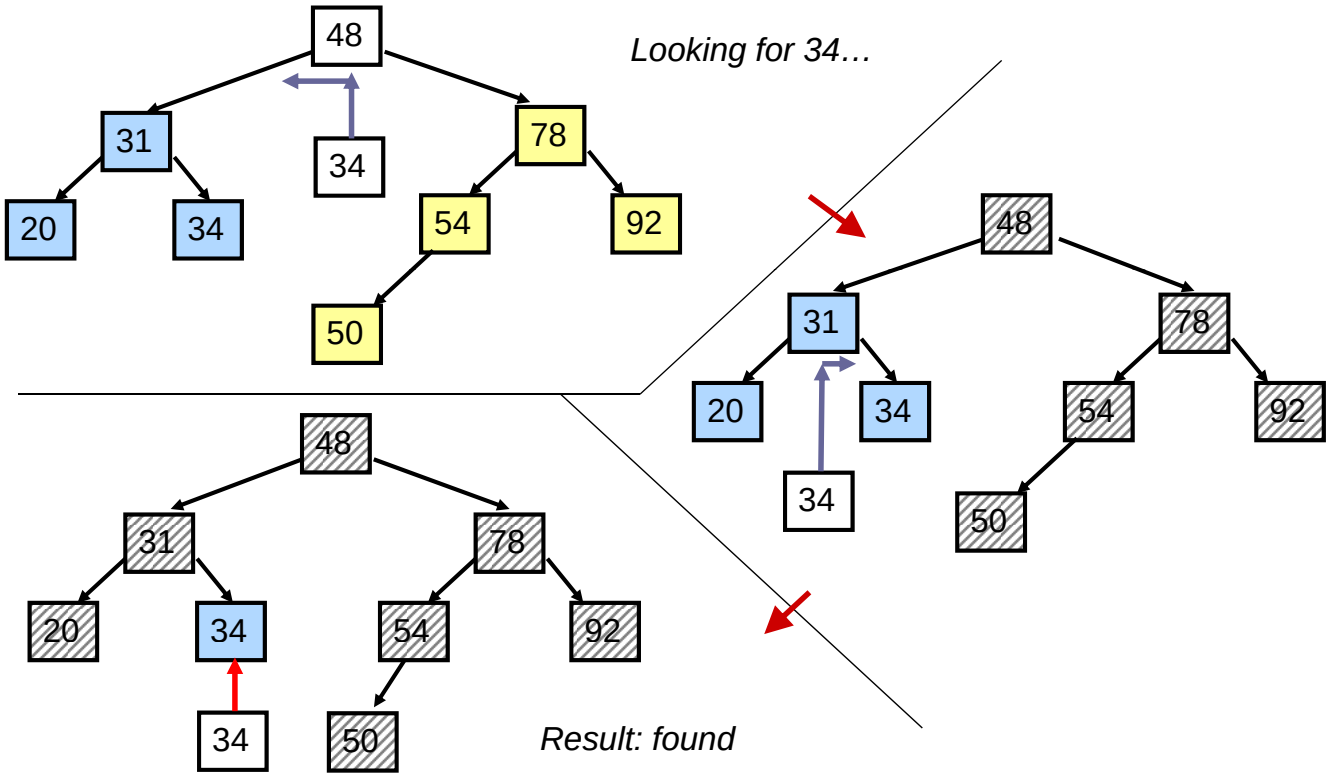
Binary Search Tree: Find

Recall the binary search algorithm, and apply it here:



To search for **X** in a tree with root **R**,
compare **X** with the data value $\mathbf{R_v}$ of **R**
if **X** = $\mathbf{R_v}$ success - terminate
else if X < $\mathbf{R_v}$ make the root of the left subtree of **R** the new **R**
else if X > $\mathbf{R_v}$ make the root of the right subtree of **R** the new **R**
Repeat the above until success or finish.
The binary search finishes without success when **R** is null (cannot go left or right)

Binary Search Tree: Find (an example)



Binary Search Tree ADT: Find Code

```
protected Comparable find(Comparable x, BinaryNode r)
                                        throws ItemNotFound
{
   while (r != null)
   {
      if (x.compareTo (r.data) < 0)
         r = r.left;
      else if (x.compareTo (r.data) > 0)
         r = r.right;
      else
         return r.data;
   }

   throw new ItemNotFound("Find fails");
}
```

Iterate through the branches.

Move through the tree as necessary.

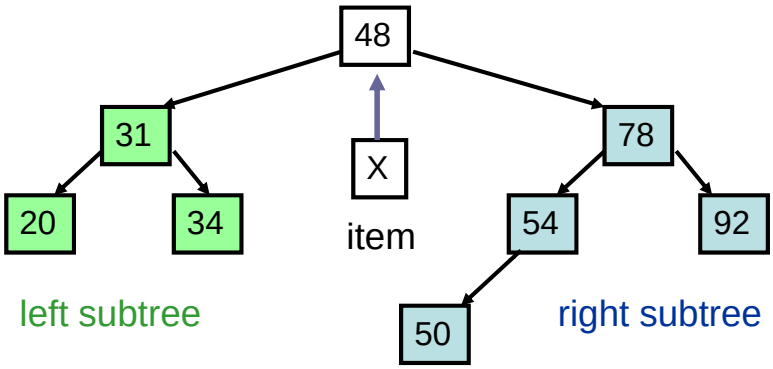
Throw an exception if the find operation failed.

The throwing of exceptions at a find failure is purely a design decision – you could, for example, make it return a null reference instead*. However what if data is meant to be null? Returning node instead of data avoids this problem.

Binary Search Tree: Insert

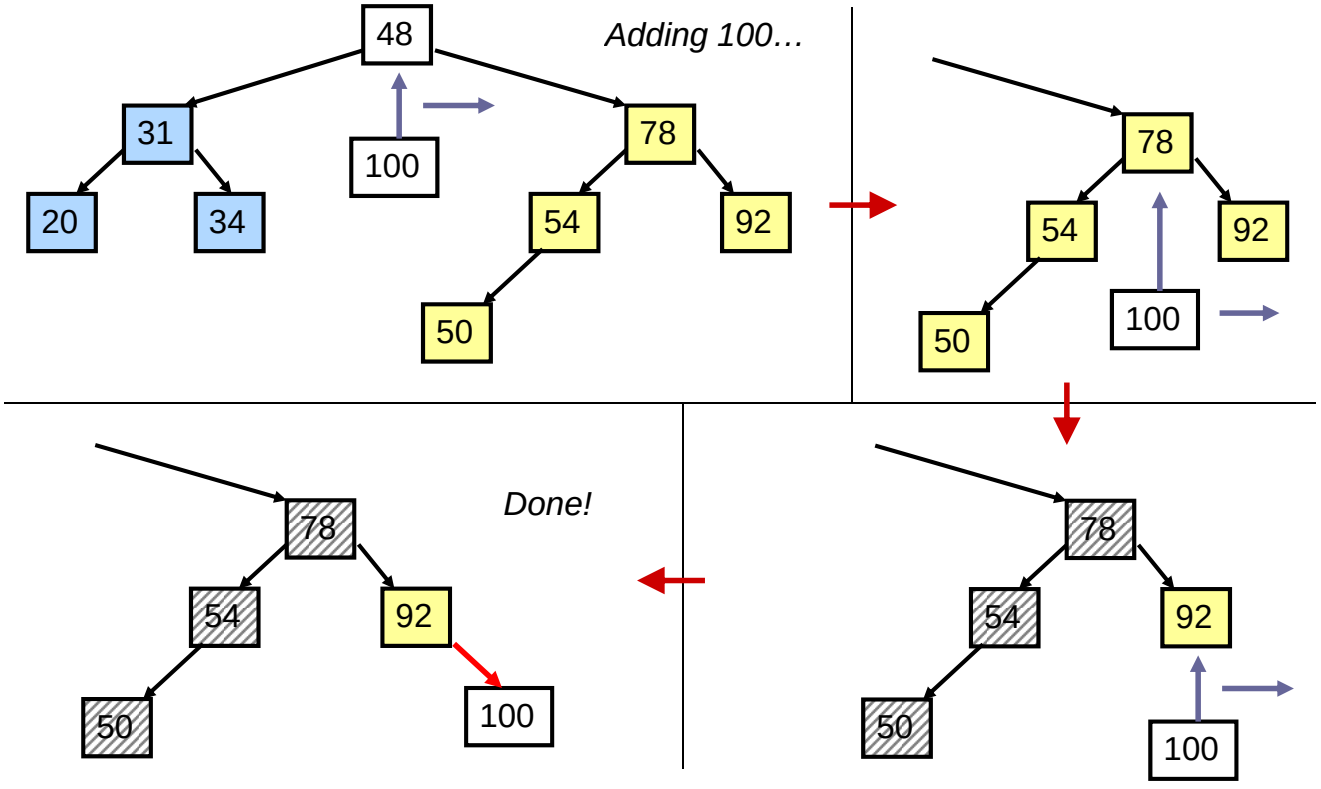
Insertion can be done recursively:

(Note how simple the code ends up being…)



To insert a value **X** into a tree with root **R** that stores a data value $\mathbf{R_v}$
if the tree is empty simply set its root **R** to the new node – recursion returns
else if $\mathbf{X} < \mathbf{R_v}$, insert the value in the left subtree
else if $\mathbf{X} > \mathbf{R_v}$, insert the value in the right subtree
else if $\mathbf{X} = \mathbf{R_v}$, throw a duplicate item exception – recursion returns
Recursion returns when either the new node has been inserted or a duplicate is found

Binary Search Tree: Insert (an example)



Binary Search Tree ADT: Insert Code

```
protected BinaryNode insert(Comparable x, BinaryNode r)
                                        throws DuplicateItem
{
   if (r == null)
      r = new BinaryNode (x);
   else if (x.compareTo (r.data) < 0)
      r.left = insert (x, r.left);
   else if (x.compareTo (r.data) > 0)
      r.right = insert (x, r.right);
   else
      throw new DuplicateItem("Duplicate item attempt");

   return r;
}
```

We create a new node, to save the existing root node given (remember, r is just a local parameter variable)

Move through the tree if necessary.

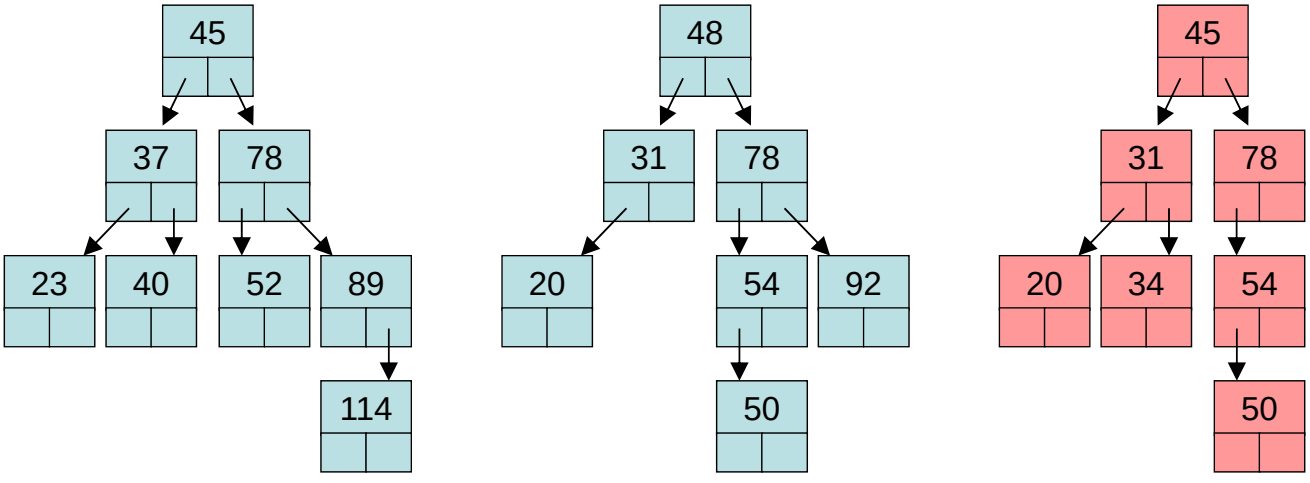
We need to return the new node reference.

In the publically-available insert method, the root is changed by assigning the protected insert's return value to it.

Balanced Binary Tree

A binary tree is balanced if and only if, for every node, the height of its left and right subtree differ by at most 1.

Some trees are self-balancing, e.g. AVL trees (inventors Adelson, Velskii, and Landis) and red-black trees (used in java.util.TreeMap/TreeSet).



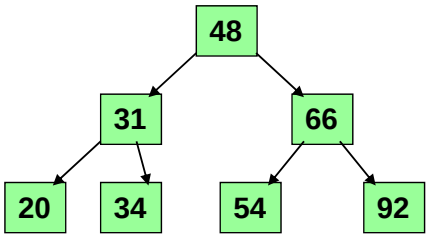
balanced tree *balanced tree* *unbalanced tree*

Binary Search Tree: Find and Insert Performance

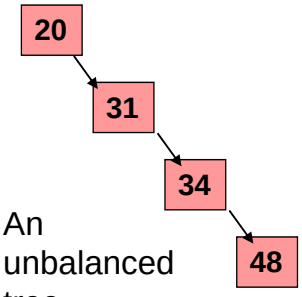
• If the tree is perfectly balanced, the situation is exactly like in binary search. A tree of **N** = 2^x nodes takes maximum **x** comparisons. That is, we have logarithmic access cost, **O(logN)**

• If the tree is unbalanced, the situation tends towards linear search. The worst case is when the tree degenerates to a mere linked list. In this case a tree of **N** nodes takes maximum **N** comparisons. That is, we have linear access cost, **O(N)**

• Binary search tree is worst to store sorted data values and is best to store random data values.



A balanced tree



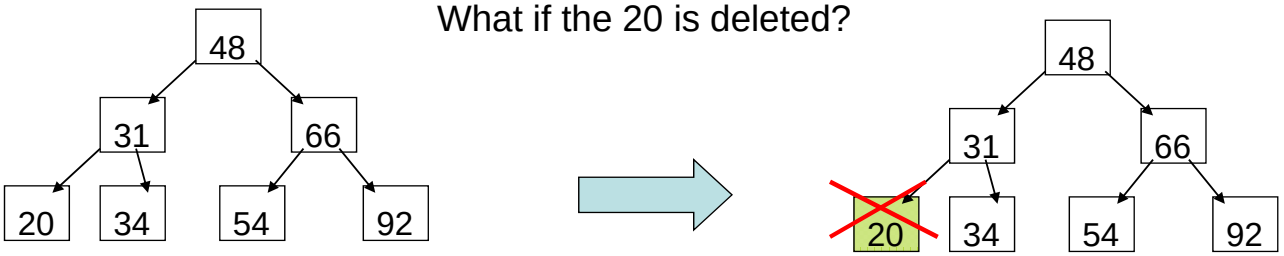
An unbalanced tree

Binary Search Tree: Deletion (1)

Removing an element from a BST can be broken down into 3 cases.

Case 1: If the node to be removed is a leaf (*i.e. no children*), it can be deleted immediately.

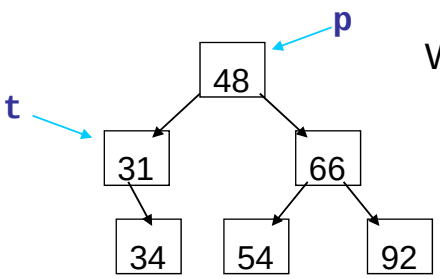
What if the 20 is deleted?



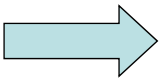
Binary Search Tree: Deletion (2)

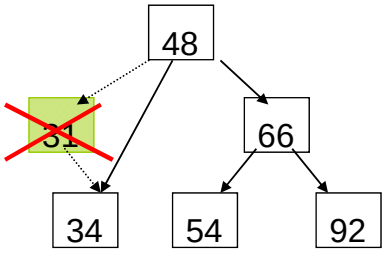
Case 2: If the node to be removed has one child…

the node can be deleted by altering the link from its parent to itself, to point to its child.



What if the 31 is deleted?





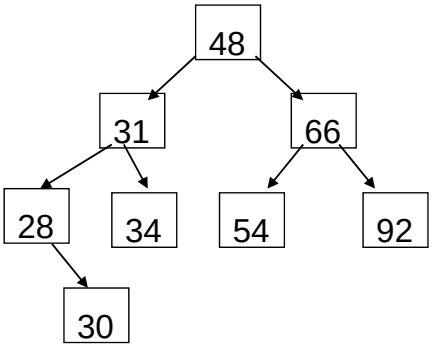

```
p.left = t.right
```


Binary Search Tree: Deletion (3)

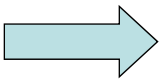
Case 3: If the node to be removed has two children…

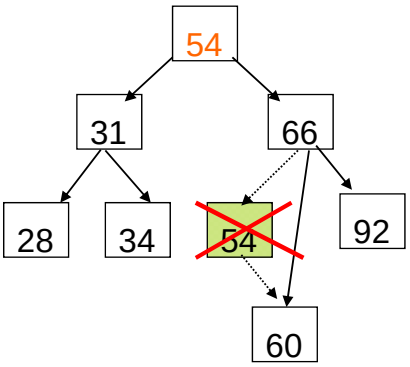
One way to remove it is by replacing the data of this node with the smallest data of the **right** sub-tree.

Then the node with the smallest data is removed (since the smallest data node can only have at most one node it can be handled as case 1 or case 2).



What if the 48 is deleted?





Binary Search Tree: Deletion Code (1)

```
protected BinaryNode remove (Comparable x, BinaryNode r)
                                          throws ItemNotFound
{
   if (r == null)
      throw new ItemNotFound("target not found to delete");

   if (x.compareTo(r.data) < 0)
      r.left = remove(x, r.left);
   else if (x.compareTo(r.data) > 0)
      r.right = remove(x, r.right);

   else if(r.left != null && r.right != null)
   {
      r.data = (findMin( r.right )).data;
      r.right = remove( r.data, r.right);
   }
```

Find the target node to delete, moving through the tree as necessary

Deletion process if there are two children

Binary Search Tree: Deletion Code (2)

```
   else
   {
      if (r.left != null)
         r = r.left;
      else
         r = r.right;
   }
   return r;
}

BinaryNode findMin (BinaryNode r)
{
   if (r == null)
      return null;
   else if (r.left == null)
      return r;

   return findMin (r.left);
}
```

Deletion process if there is at most one child node.

Find the smallest data in a tree by moving as far left as possible.

Traversing a Binary Tree

Traversing a tree means systematically processing every node in that tree.

For a binary tree, there are two links for each node; therefore, we have three basic orders in which we can process the tree:

- *preorder:* process the node, then visit the left and right subtrees
- *inorder:* visit the left subtree, then process the node, and then the right subtree
- *postorder:* visit the left and right subtrees, and then process the node

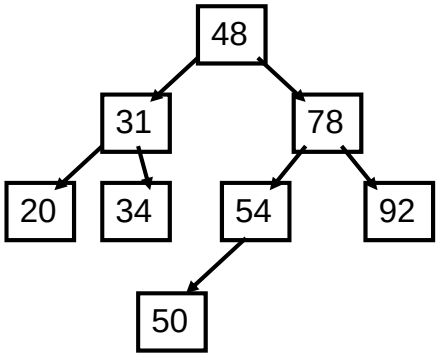
Every node is accessed, therefore performance is **O(*N*)**

Traversal: Pre-Order

```
protected void preorder (BinaryNode r)
{
   if (r == null) return;
   System.out.print (r.data + " ");
   preorder (r.left);
   preorder (r.right);
}
```

Output: 48 31 20 34 78 54 50 92

Note: The root is at the start.

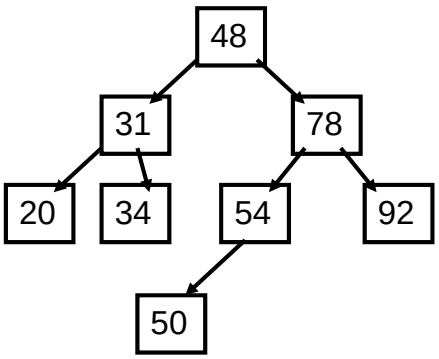


Traversal: In-Order

```
protected void inorder(BinaryNode r)
{
   if (r == null)
      return;

   inorder (r.left);
   System.out.print (r.data + " ");
   inorder (r.right);
}
```

Output: 20 31 34 48 50 54 78 92

Note: output is sorted for binary search tree



Traversal: Post-Order

```
protected void postorder(BinaryNode r)
{
   if (r == null)
      return;

   postorder (r.left);
   postorder (r.right);
   System.out.print (r.data + " ");
}
```

Output: 20 34 31 50 54 92 78 48

Note: The root is at the end